The beauty of bootstrapping and the joy of JIT

Michael Spivey

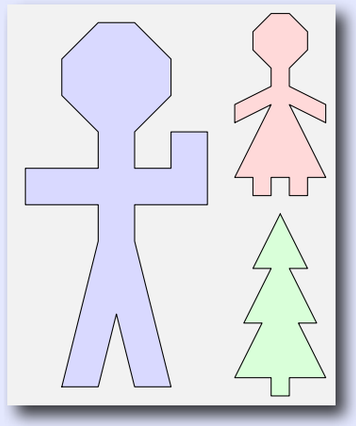
Department of
COMPUTER
SCIENCE

The message

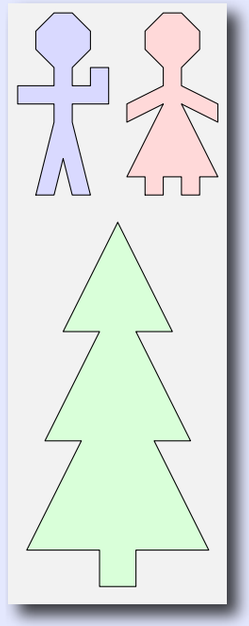
- Implementing your own language is not too difficult and a lot of fun.
- Decent performance is quite easy to achieve.

GeomLab: an algebra of pictures

man $ (woman & tree)



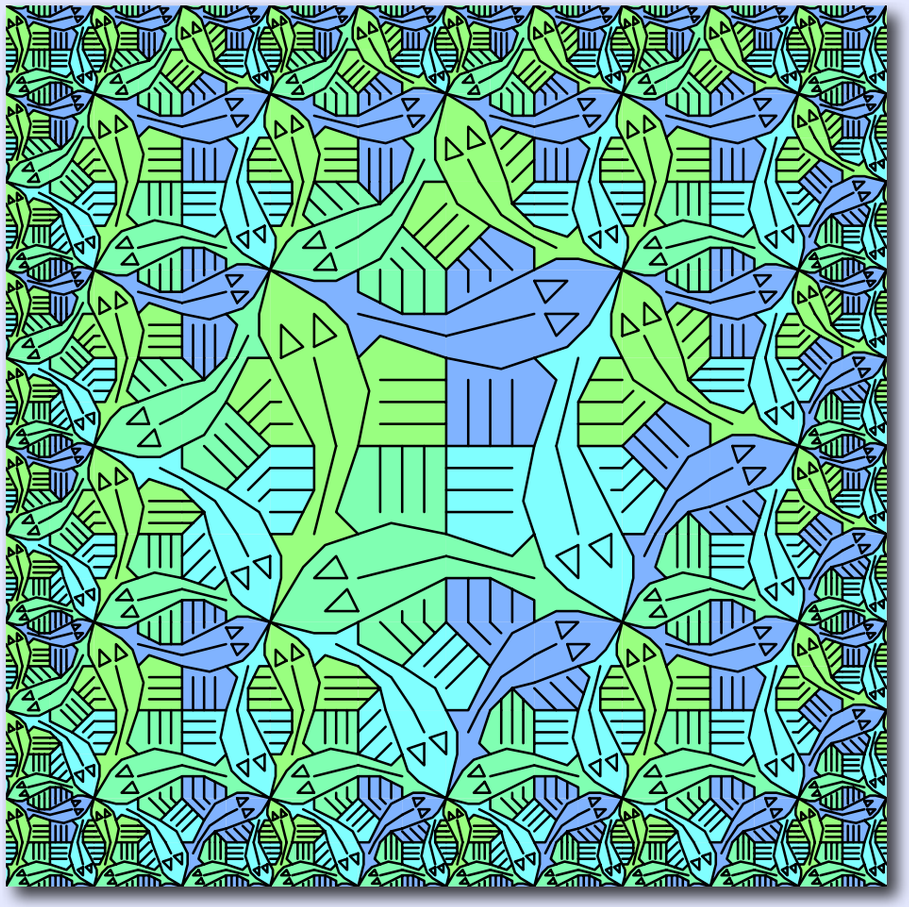
(man $ woman) & tree



The big challenge

Use recursion to describe this Escher picture:

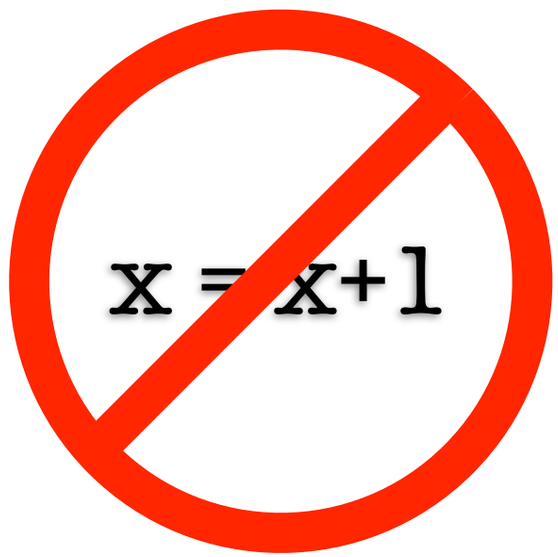
frame(corner(2), side(2), U)



What's programming like?

The old-fashioned view:

- A program is a list of instructions.
- Algorithms are designed with flowcharts.
- Computer memory is boxes containing numbers.
- It's all ones and zeros!

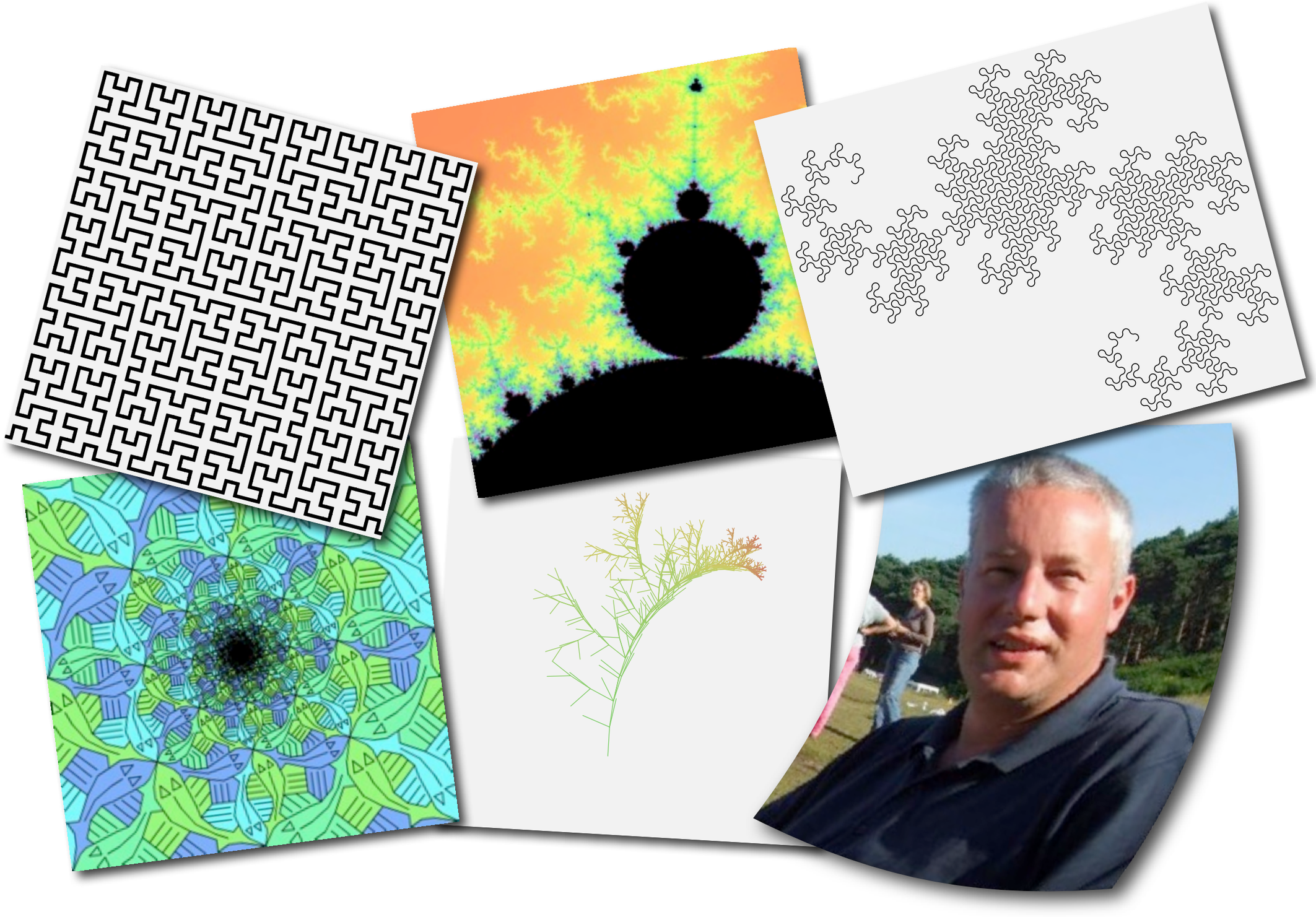


What's it really like?

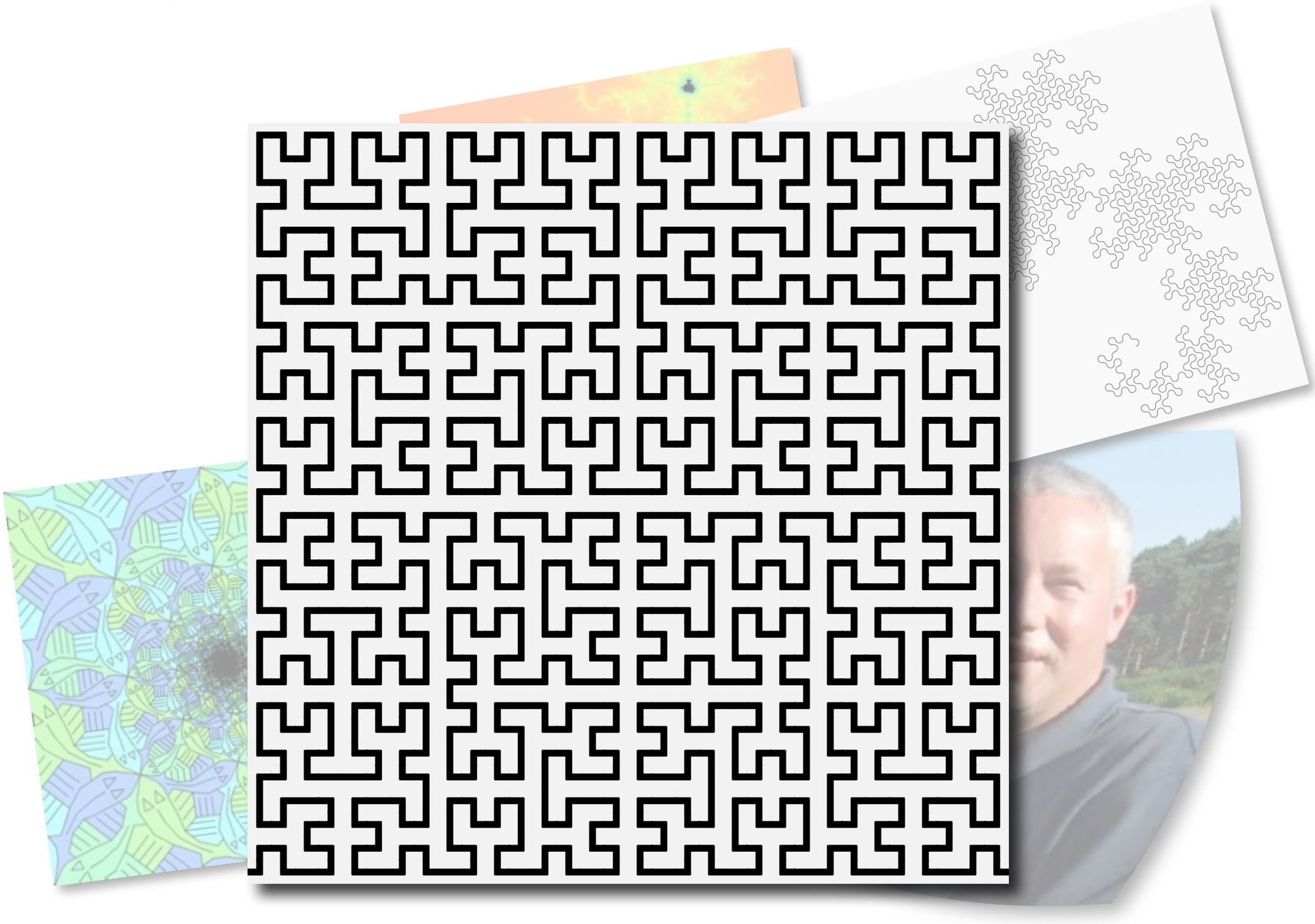
A modern view:

- A program is an artificial world.
- Objects behave according to precise rules.
- Combining simple behaviours can let a complex whole emerge.
- We can use things without knowing how they are made.

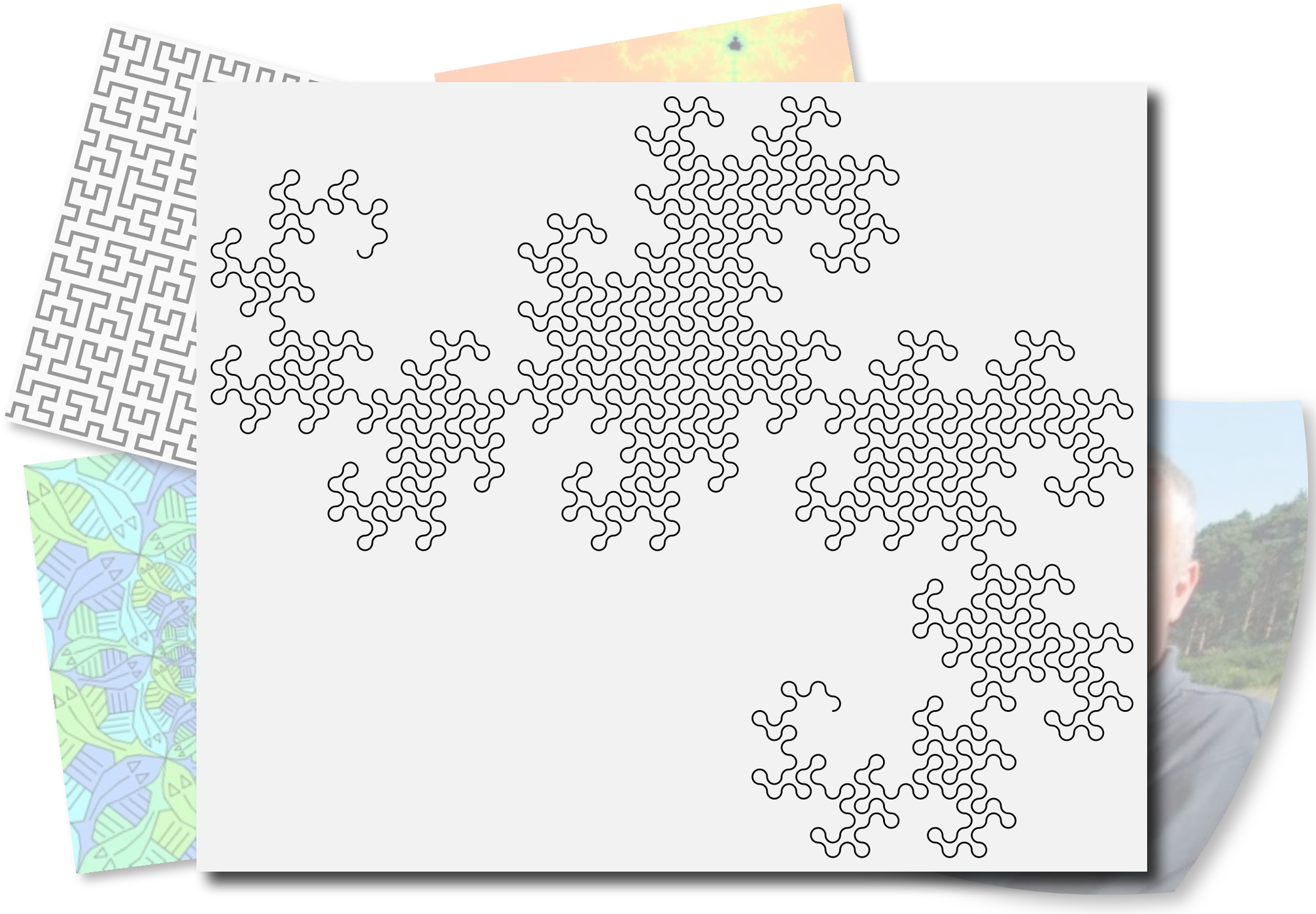
But we'd like to do more!



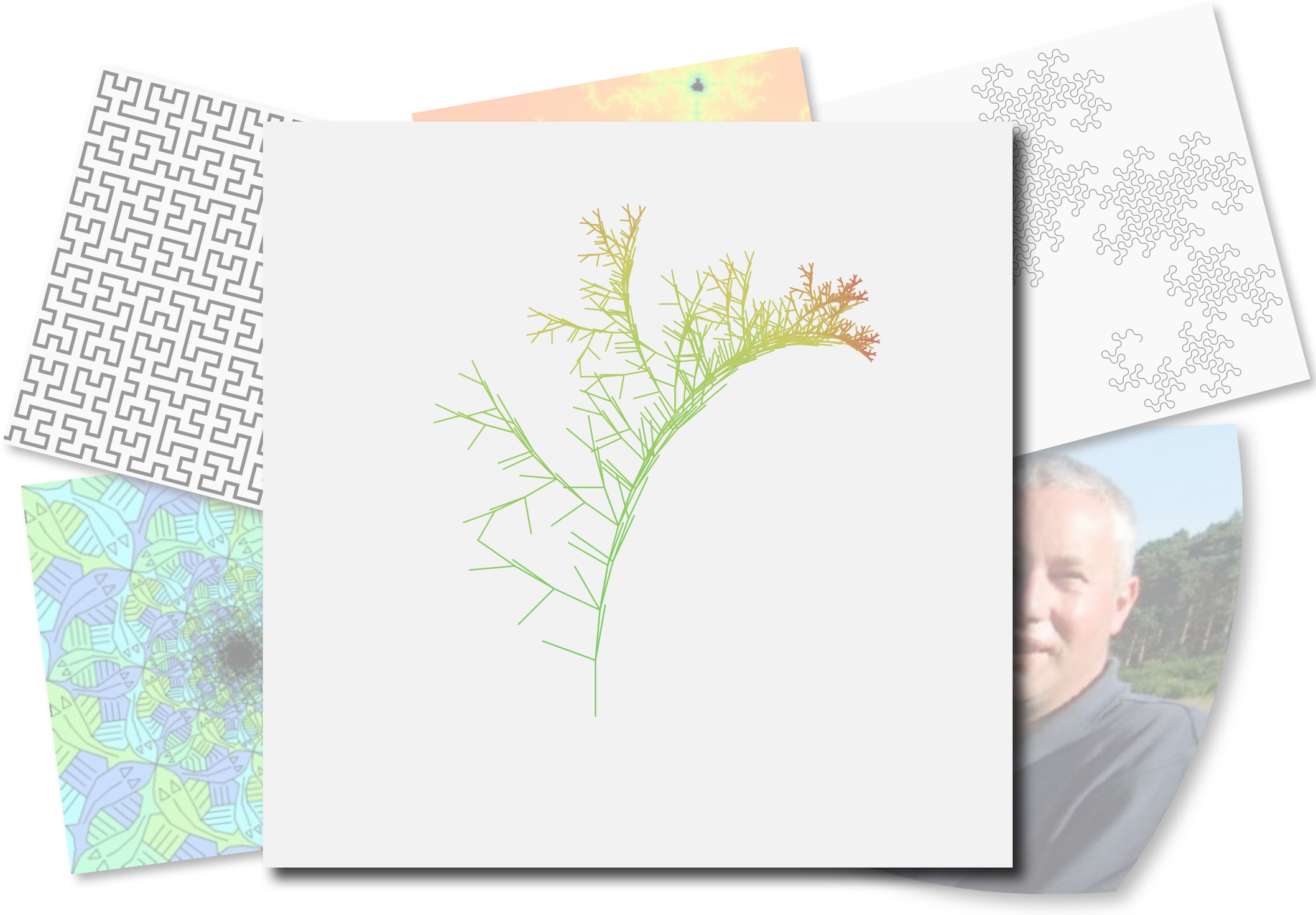
Tiling patterns



Turtle graphics



Fractal plants



Pixel images

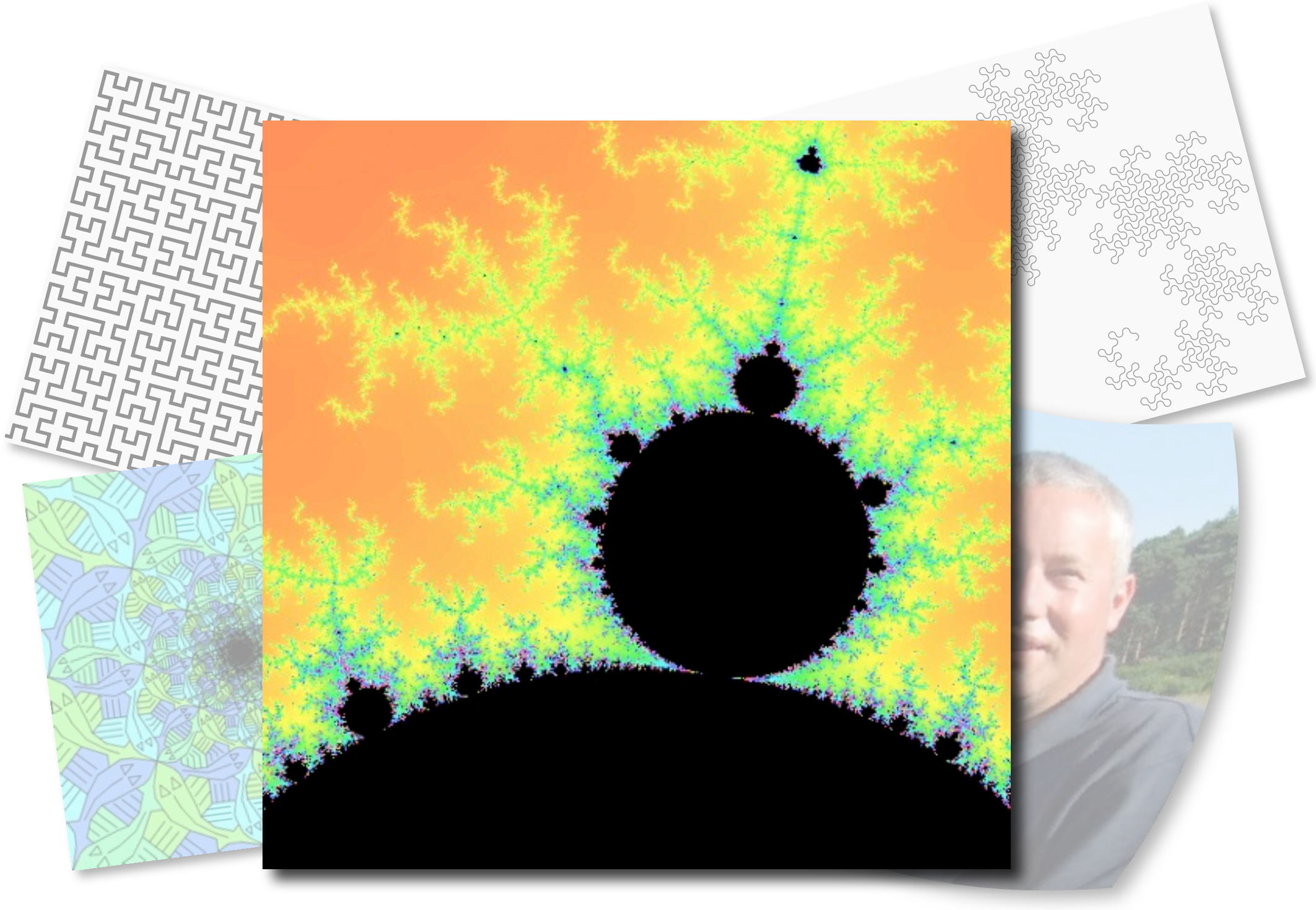
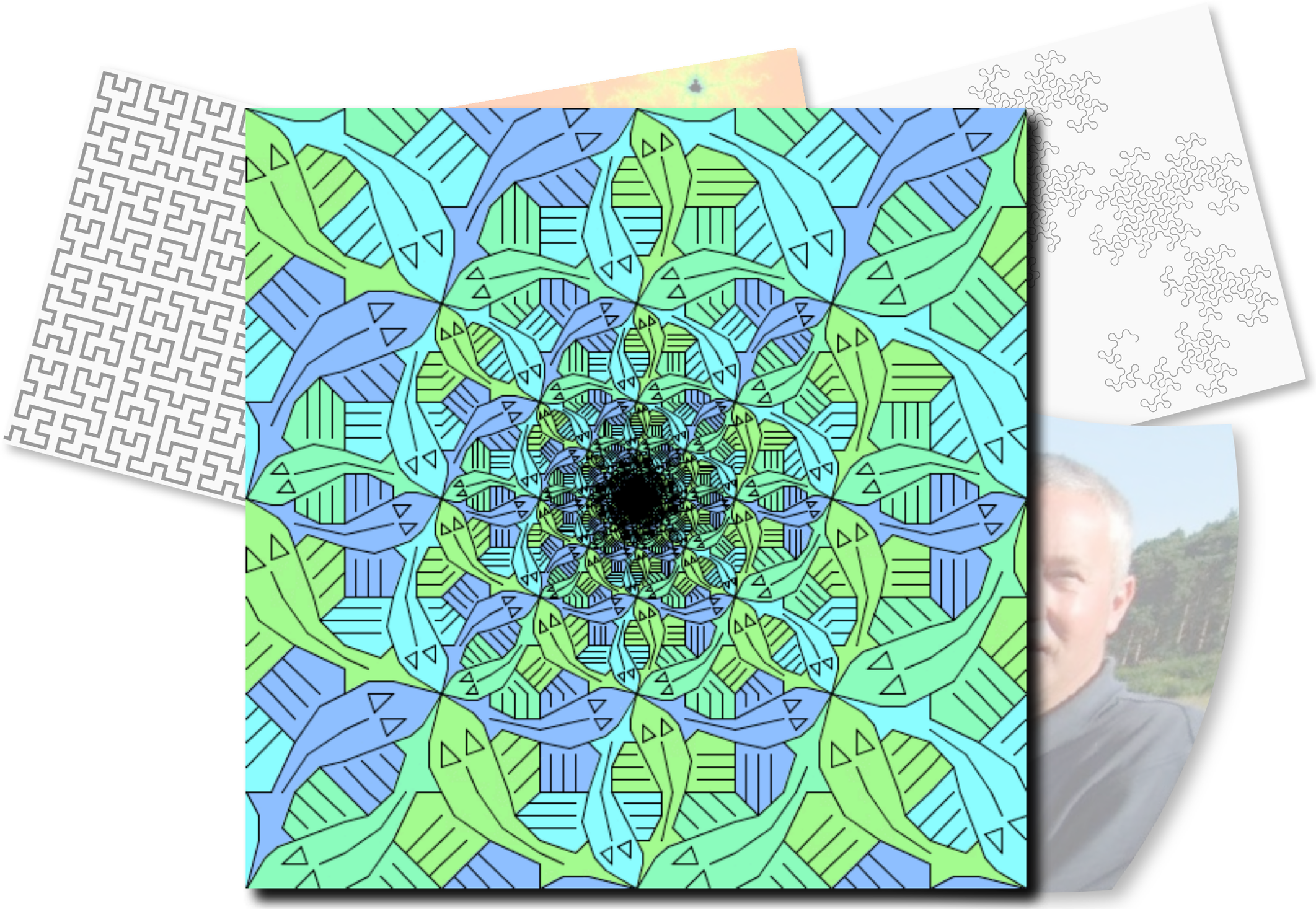


Photo manipulation

Animations



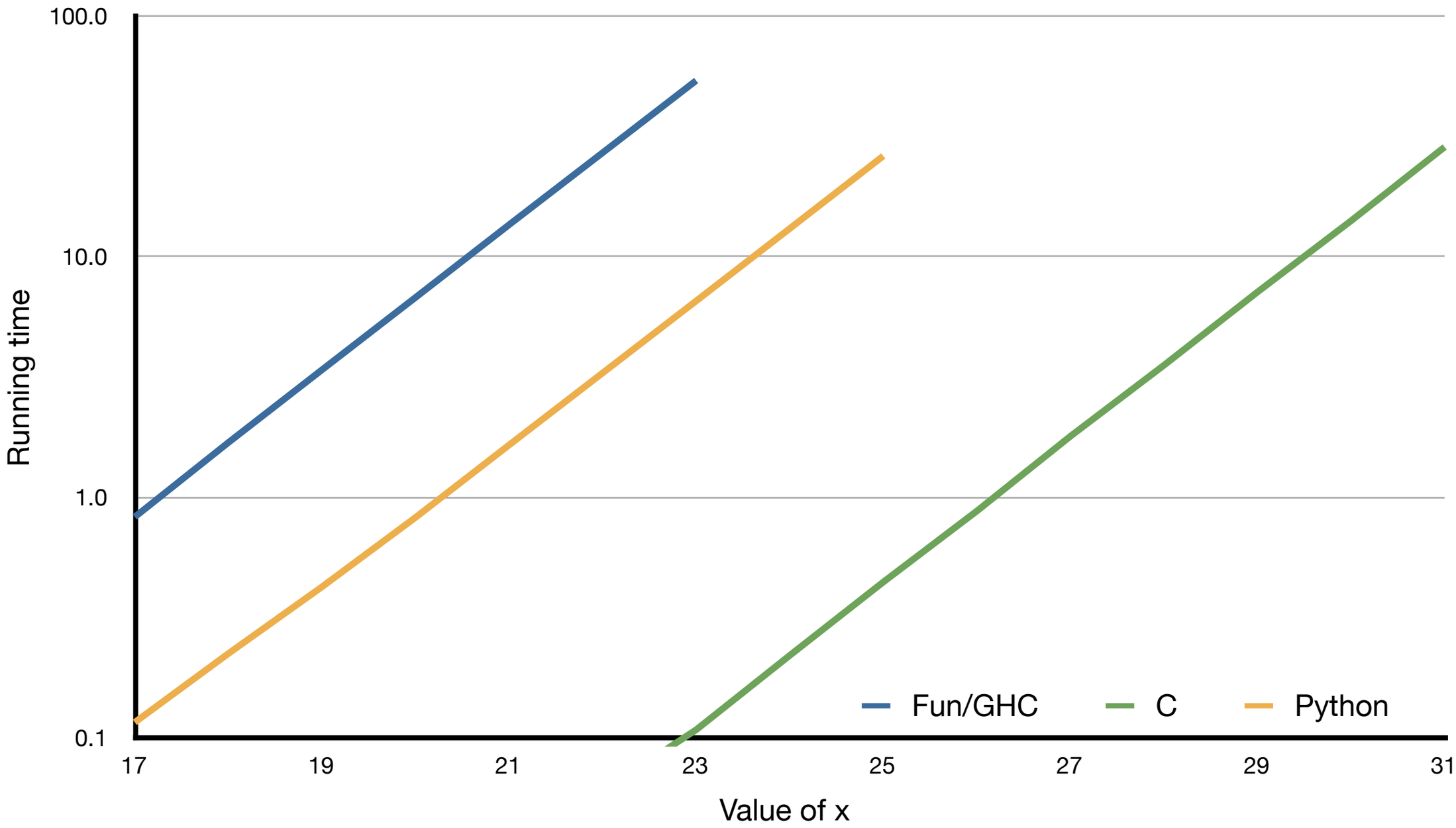
A first implementation

Parse the program to form an abstract syntax tree, then use a tree-walking interpreter.

- Not hard to do, but a bit heavy to write in Java.
- Performance is good enough for the tiling patterns, because all the intensive graphics is done in Java directly.
- But it runs out of steam if we try to compute a picture pixel-by-pixel.

The performance gap

define f(*O*) = 1 | f(n+1) = f(n) + f(n) in f(x)



Closing the gap – portably

If (higher-order) functions can be represented in Java directly, we could remove the interpreter overhead.

A typical functional program:

```
define inc(xs, y) = map(lambda (x) x + y, xs)
```

How can we represent it as objects?

Closures as objects

define inc(xs, y) = map(lambda (x) x + y, xs)

```
class InnerLambda {
    private Name plus = Name.find(“+”);
    private Value y;

    public InnerLambda(Value y) { this.y = y; }

    public Value apply1(Value x) {
        return plus.glodef.apply2(x, y);
    }
}

class IncFunction {
    private Value map = Name.find(“map”);

    public Value apply2(Value xs, Value y) {
        return map.glodef.apply2(new InnerLambda(y), xs);
    }
}
```

Compiling via Java

We could translate a GeomLab program to a Java program, then compile with the Java compiler.

- The Java JIT would then give us something like native code performance.
- But we want something more dynamic: type expressions at a prompt, and have them evaluated immediately at high speed.

Be more dynamic!

Build a compiler from the GeomLab language to JVM bytecode.

Each function definition becomes a new class that is loaded into the running program.

- The compiler produces a binary class file in exactly the same format as the Java compiler.
- Then it's passed to the class loader and becomes part of the program.

Java's JIT translator still ought to give us high performance.

Bootstrapping the compiler

How shall we write the compiler?

- In Java? Yuck!
- In GeomLab? Is that possible?

Yes! Use the interpreter to let the compiler compile itself.

Save the object code from one version of the compiler and use it to compile the next version.

Compiler phases

- Lexer (350 lines, Java) + Parser (270 lines, Fun) build abstract syntax tree.
- Front end (320 lines, Fun) generates FunCode.
- Back end (930 lines, Java) + Bytecode assembler (1500 lines, Java) generate bytecode.
- JIT (part of the JVM) generates native code.

The bootstrap process

Given a compiler *A*, *a* that generates poor code and runs slowly:

Rewrite it to make a compiler *B* that generates better code.

- Use *a* to compile *B*, giving $b_1 = a(B)$, a slow compiler that generates better code.
- Then use b_1 to compile *B*, giving $b_2 = b_1(B)$, a faster compiler that generates better code.
- Check that $b_2(B) = b_1(B)$.

Abstract syntax tree

The abstract syntax tree of a program is represented as a value in the language:

define inc(xs, y) = map(*lambda* (x) x + y, xs)

```
[#fun, #inc, 2,
   [[[[#var, #xs], [#var, #y]],
      [#apply, [#var, #map],
         [#lambda, [[#var, #x]],
            [#apply, [#var, #+],
               [#var, #x], [#var, #y]]],
         [#var, #xs]]]]]
```

First translation: into FunCode

define inc(xs, y) = map(lambda (x) x + y, xs)

```
inc:
    [#GLOBAL, #map]
    [#PREP, 2]
    [#QUOTE, <lambda>]
    [#ARG, 1]
    [#CLOSURE, 1]
    [#ARG, 0]
    [#CALL, 2]
    [#RETURN]
```

```
<lambda>:
    [#GLOBAL, #+]
    [#PREP, 2]
    [#ARG, 0]
    [#FVAR, 1]
    [#CALL, 2]
    [#RETURN]
```

Second translation: into JVM

define inc(xs, y) = map(lambda (x) x + y, xs)

```
inc.apply2:
     ALOAD 0                                          [#GLOBAL, #map]
     GETFIELD funjit/JitFunction.consts
     ICONST 0
     AALOAD
     CHECKCAST funbase/Name
     GETFIELD funbase/Name.glodef
     GETFIELD funbase/Value.subr                      [#PREP, 2]
     ALOAD 0
     GETFIELD funjit/JitFunction.consts               [#QUOTE, <lambda>]
     ICONST 1
     AALOAD
     ALOAD 2                                          [#ARG, 1]
     ICONST 2                                         [#CLOSURE, 1]
     ANEWARRAY funbase/Value
     DUP_X1
     SWAP
     ICONST 1
     SWAP
     AASTORE
     INVOKEVIRTUAL funbase/Value.makeClosure
     ALOAD 1                                          [#ARG, 0]
     INVOKEVIRTUAL funbase/Function.apply2            [#CALL, 2]
     ARETURN                                          [#RETURN]
```

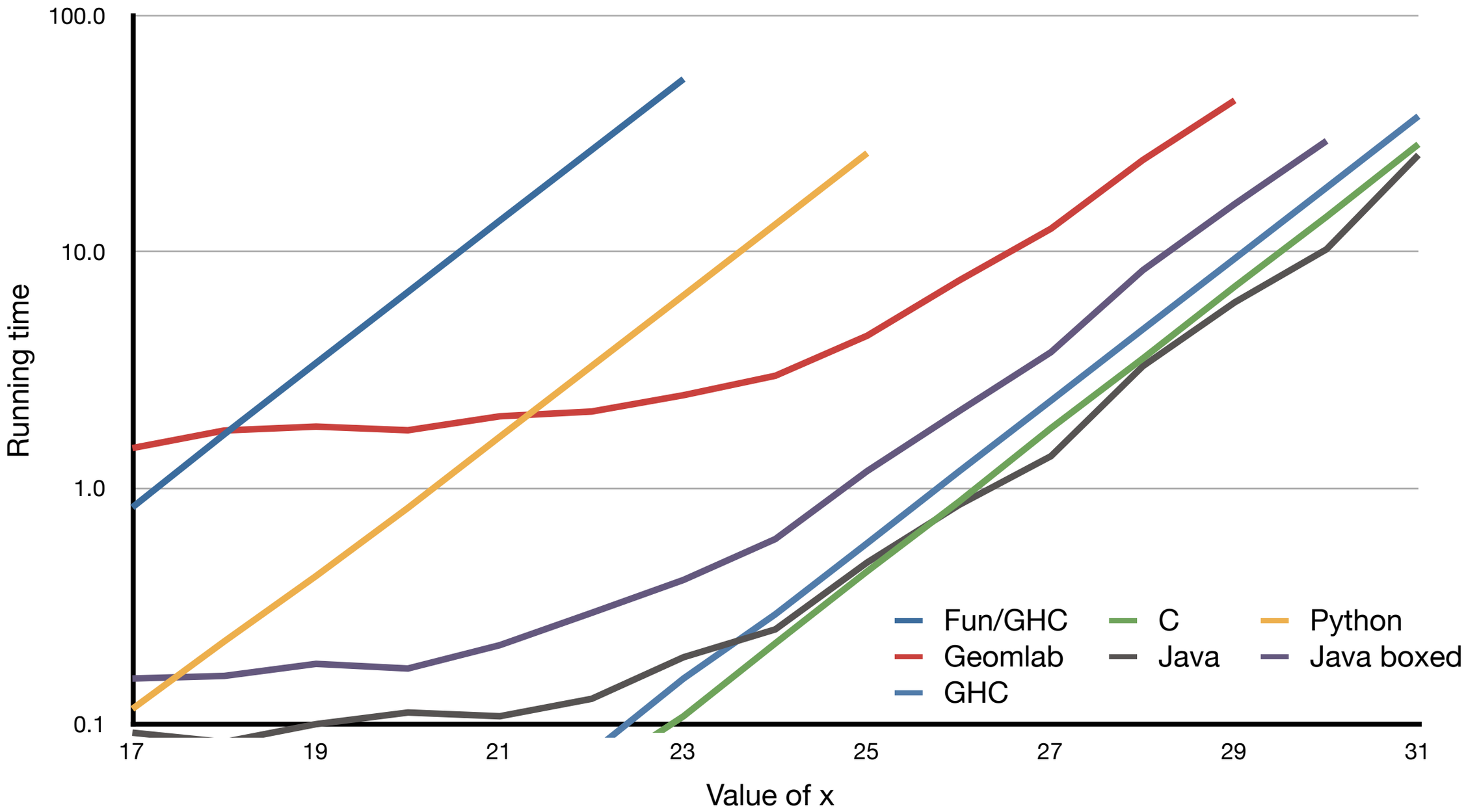
About the implementation

- Eager
- Only direct tail-recursion
- Uses Java stack
- Best to avoid the reflection API

- ASM library is recommended
- Cf: Scala, Clojure, Jython

How did we do?

define f(*O*) = 1 | f(n+1) = f(n) + f(n) *in* f(x)



Not too complicated

Whole GeomLab system is 10 000 lines of Java + 1200 lines in the GeomLab language.

Java code evenly divided among basis for functional language, JIT translator, GUI and graphics library.

Implementing your own language is an entirely feasible project.